

# Imprimer avec Python

*Nous avons beaucoup travaillé jusqu'ici pour apprendre à développer des applications véritablement utilisables. Afin de progresser encore dans cette voie, il nous reste à découvrir comment générer, avec ces mêmes applications, des documents imprimables de qualité professionnelle. Cela implique que nous devons être capables de programmer la réalisation de pages imprimées contenant du texte, des images et des dessins...*

## Un peu d'histoire

Lorsque les premiers ordinateurs sont apparus au milieu du siècle dernier, il a fallu rapidement imaginer des procédés techniques pour imprimer sur papier les résultats produits par les programmes informatiques. À l'époque de ces premières machines, en effet, les écrans n'existaient pas encore, et les méthodes de stockage de l'information étaient par ailleurs très rudimentaires et très coûteuses. On inventa donc les premières imprimantes (en adaptant les « téléscripteurs » contemporains) afin de pouvoir non seulement consulter, mais aussi archiver les données produites.

Ces premières machines n'imprimaient au début que des chiffres et des caractères majuscules non accentués, ce qui pouvait suffire en ces temps où les ordinateurs n'étaient utilisés que pour la résolution de problèmes scientifiques et la comptabilité de grosses entreprises.

Cette situation n'a pas beaucoup changé jusqu'à l'apparition des premiers ordinateurs personnels : au début des années quatre-vingt, en effet, les premières imprimantes « grand public » ne pouvaient toujours imprimer que du texte. Les langages de programmation de l'époque étaient alors capables de gérer l'impression sur papier à l'aide de quelques instructions simples : il suffisait en effet de pouvoir expédier à ces imprimantes les suites de caractères à imprimer, entrecoupées de quelques codes « non imprimables » réservés à des commandes particulières : tabulations, fins de lignes, fins de pages, etc.

Ces instructions simples, telles par exemple l'instruction **LPRINT** du langage BASIC, s'utilisaient pratiquement de la même manière que celles qui permettent d'afficher des caractères à l'écran, ou d'enregistrer des lignes de texte dans un fichier (comme la fonction **print()** ou la méthode **write()** de Python).

Les imprimantes modernes sont très diverses et performantes : la plupart sont désormais des *imprimantes graphiques*, qui n'impriment plus des suites de caractères typographiques prédéfinis dans une police unique, mais plutôt des combinaisons quelconques de points minuscules de différentes couleurs. Il n'existe donc plus de restriction à ce qu'une telle imprimante peut représenter : alignements divers de caractères typographiques tirés de n'importe quel alphabet, dans une variété infinie de polices et de tailles différentes, mais aussi images et dessins de toute sorte, voire photographies.

C'est évidemment tout bénéfice pour l'utilisateur final, puisqu'il peut accéder à une gamme de possibilités créatives immense, qu'aucun imprimeur n'aurait osé rêver jadis. Pour le programmeur en revanche, cela signifie d'abord une complexité bien plus grande.

Pour commander l'impression d'un texte à une imprimante moderne, il ne suffit plus de lui envoyer la suite de caractères du texte, comme on le ferait pour enregistrer ce même texte dans un fichier. Il faut désormais considérer que chaque caractère est un petit dessin que l'imprimante devra reproduire point par point en un endroit précis de la page. Un programme informatique destiné à produire des rapports imprimés doit donc être doté entre autres choses d'une ou plusieurs bibliothèques de *glyphes*<sup>1</sup> correspondant aux polices et aux styles que

<sup>1</sup>On appelle *glyphe* le dessin d'un caractère typographique (voir aussi page Erreur : source de la référence non trouvée).

l'on souhaite utiliser, ainsi que d'algorithmes efficaces pour traduire ces glyphes en matrices de points, avec une résolution bien déterminée et dans une certaine taille.

Pour transmettre toutes ces informations, il faudra en outre disposer d'un véritable *langage* spécifique. Comme vous pouvez vous y attendre, il existe un certain nombre de ces langages, avec des variantes pour piloter les différents modèles et marques d'imprimantes du marché.

## *L'interface graphique peut aider*

Au début de ce livre, nous avons brièvement expliqué qu'il existe toute une hiérarchie des langages informatiques, que ce soit pour programmer le corps d'une application, pour envoyer une requête à un serveur de bases de données, pour décrire une page web, etc. Vous savez ainsi qu'il existe des langages de bas niveau, à la syntaxe souvent obscure, laborieux à utiliser du fait qu'ils nécessitent de très nombreuses instructions pour la moindre commande, mais heureusement aussi des langages de haut niveau (comme Python), beaucoup plus agréables à utiliser grâce à leur syntaxe se rapprochant du langage humain, et plus efficaces en règle générale, du fait de leur mise en œuvre plus rapide.

Pour ce qui concerne le dialogue avec les imprimantes modernes, c'est dorénavant le système d'exploitation de l'ordinateur (au sens large) qui s'occupe de prendre en charge les *langages de bas niveau*. Ceux-ci ne sont plus guère utilisés que pour écrire des programmes interpréteurs particuliers appelés *pilotes d'imprimantes*, lesquels sont généralement fournis par les constructeurs de ces imprimantes, ainsi que des bibliothèques logicielles qui assureront l'interfaçage avec les langages de programmation plus généralistes.

Pour des raisons de cohérence évidentes, les bibliothèques logicielles permettant de créer ces applications avec interface graphique qui font apparaître des images de toutes sortes dans des fenêtres à l'écran (textes formatés, bitmaps, etc.) sont souvent capables de gérer aussi la mise en page imprimée des mêmes images. Certaines de ces bibliothèques sont intégrées au système d'exploitation lui-même (cas des *API* de Windows), mais il en existe d'autres moins dépendantes de celui-ci, auxquelles vous devriez toujours accorder votre préférence. Ne serait-il pas dommage, en effet, de restreindre la portabilité de vos scripts Python ?

Des bibliothèques d'interfaçage graphique sont donc disponibles pour votre langage de programmation. Elles vous proposent des classes d'objets permettant de construire des pages à imprimer à l'aide de widgets, un peu comme lorsque vous construisez des fenêtres à l'écran. Avec *Tkinter*, par exemple, vous pouvez préparer le dessin d'une page à imprimer dans un canevas, puis expédier une représentation de celui-ci à une imprimante – à la condition que cette imprimante « comprenne » le langage d'impression *PostScript* (malheureusement peu répandu). D'autres bibliothèques d'interfaçage telles *WxPython* ou *PyQt* proposent davantage de possibilités.

Le principal intérêt des techniques d'impression exploitant les bibliothèques d'interfaces graphiques réside dans le fait que le script Python que vous écrivez sur cette base contrôle tout : vous pouvez donc faire en sorte que la page à imprimer soit construite morceau par morceau à l'écran par l'utilisateur, la lui montrer à tout moment telle qu'elle apparaîtra imprimée, et déclencher l'impression elle-même de la manière qui vous convient le mieux.

Ces techniques présentent cependant quelques inconvénients : leur portabilité n'est pas toujours effective sur tous les systèmes d'exploitation, et leur mise en œuvre nécessite l'apprentissage de concepts quelque peu rébarbatifs (*device context*, etc.). De plus, ce type d'approche montre vite ses limites lorsque l'on envisage de produire des documents imprimés d'un certain volume, avec des textes comportant de multiples paragraphes, entrecoupés de figures, dont on souhaite contrôler finement la mise en page, avec des variations de style, des espacements, des tabulations, etc.

## *Le PDF, langage de description de page pour l'impression*

En fait, ces limitations sont liées à une problématique que nous avons déjà rencontrée : si nous utilisons ces techniques, le niveau trop bas du langage utilisé pour dialoguer avec l'imprimante risque de nous obliger à

réinventer dans nos scripts des mécanismes complexes, alors que ceux-ci ont très certainement déjà été imaginés, testés et perfectionnés par des équipes de développeurs compétents.

*Lorsqu'au chapitre 16 nous avons abordé la gestion des bases de données, par exemple, nous avons vu qu'il était préférable de confier l'essentiel de cette tâche compliquée à un système logiciel spécialisé (le moteur de base de données, SGBDR), plutôt que de vouloir tout réinventer dans nos scripts. Ceux-ci peuvent se contenter de générer des instructions d'un langage de haut niveau (le SQL) parfaitement adapté à la description des requêtes les plus complexes, et laisser au SGBDR le soin de les décortiquer et de les exécuter au mieux.*

D'une manière légèrement différente, nous allons vous montrer dans les pages qui suivent que vous pouvez aisément construire avec Python les instructions d'un **langage de haut niveau** mis au point pour décrire complètement et dans ses moindres détails ce qu'une imprimante doit faire apparaître sur la page imprimée. Ce langage est le **PDF** (*Portable Document Format*).

*Conçu à l'origine par la société Adobe Systems en 1993 pour décrire les documents à imprimer d'une manière totalement indépendante de tout logiciel, matériel et/ou système d'exploitation, le **PDF** est souvent présenté plutôt comme un format de fichier, parce qu'il est habituellement généré par des bibliothèques de fonctions ou des logiciels spécialisés sous forme de scripts complets, plutôt qu'édité tel quel. Cela dit, c'est bel et bien un langage de description de pages très élaboré, qui a acquis le statut de standard quasi universel. Au départ propriétaire, ce format est devenu un standard ouvert en 2008. N'ayez aucune crainte quant à sa pérennité ou à la liberté de l'utiliser avec vos propres applications.*

Un document PDF est donc un script décrivant les textes, les polices, les styles, les objets graphiques et la mise en forme d'un ensemble de pages imprimables, lesquelles devront être restituées de façon identique, quelles que soient l'application et la plate-forme utilisées pour le lire.

Une particularité importante du langage PDF est qu'en général il n'est pas directement compréhensible par les imprimantes ordinaires. Pour l'interpréter, il faut faire appel à un logiciel spécialisé tel que *Acrobat Reader*, *Foxit reader*, *Sumatra reader*, *Evince*...

Il ne faut pas y voir un inconvénient, parce que ces logiciels présentent tous l'immense intérêt de permettre une *prévisualisation* du texte imprimable. Un document PDF peut donc être consulté, archivé, etc. sans qu'il soit nécessaire de l'imprimer effectivement. Tous ces logiciels sont par ailleurs gratuits, et il existe toujours au moins l'un d'entre eux dans la configuration standard de n'importe quel ordinateur moderne.

Le présent chapitre n'a donc pas pour but de vous expliquer comment vous pouvez commander directement une imprimante au départ d'un script Python. Au lieu de cela, il va vous montrer combien il est facile et efficace de produire des documents PDF d'une grande qualité à l'aide d'instructions puissantes et parfaitement lisibles. Cette approche garantira la portabilité de vos programmes tout en leur donnant accès à des fonctionnalités d'impression très étendues, lesquelles s'intégreront particulièrement bien aux applications que vous développerez pour le Web.

L'essentiel de ce dont vous avez besoin est disponible sous la forme d'un module Python distribué sur le web sous licence libre : la bibliothèque de classes **ReportLab**.

*ReportLab est en fait le nom d'une compagnie londonienne qui a développé et maintient à jour un ensemble de classes Python destinées à générer par programme des documents PDF de haute qualité. La compagnie distribue sous licence libre toutes les bibliothèques de base du système, ce qui vous autorise donc à les utiliser sans contrainte. Elle assure sa viabilité en vendant sous licence propriétaire différents programmes complémentaires pour le traitement du PDF, et en développant des applications sur mesure pour les entreprises. Mais les bibliothèques de base seront amplement suffisantes pour faire votre bonheur.*

À ce stade de nos explications, nous devrions déjà pouvoir vous proposer un exemple de petit script montrant comment importer la bibliothèque *ReportLab* et réaliser un document PDF rudimentaire. Ce n'est malheureusement pas possible tout de suite, parce qu'il nous faut d'abord résoudre un problème de taille : les seuls modules *ReportLab* disponibles à l'heure où nous écrivons ces lignes (décembre 2011) sont tous dédiés à la version 2 de Python. Il n'existe toujours pas de module *ReportLab* pour Python 3 !

## Exploitation de la bibliothèque ReportLab

Tout ceci étant posé, nous pouvons à présent écrire nos premiers scripts générateurs de documents PDF. Ces scripts seront donc écrits avec la syntaxe de Python 3, comme l'ensemble des scripts de ce livre, mais il ne pourront (provisoirement) être exécutés que sous Python 2.6 ou 2.7. Lorsque la bibliothèque *ReportLab* sera devenue disponible pour Python 3, les quelques lignes d'adaptation en début de chaque script pourront bien évidemment être supprimées.

### Un premier document PDF rudimentaire

```

1#      ### Ébauche d'un document PDF minimal construit à l'aide de ReportLab ###
2#      # Script écrit en Python 3, mais devant provisoirement être exécuté
3#      # sous Python 2.6 ou 2.7, tant que cette bibliothèque reste indisponible.
4#
5#      from __future__ import unicode_literals      # inutile sous Python 3
6#
7#      # Importation de quelques éléments de la bibliothèque ReportLab :
8#      from reportlab.pdfgen.canvas import Canvas    # classe d'objets "canevas"
9#      from reportlab.lib.units import cm            # valeur de 1 cm en points pica
10#     from reportlab.lib.pagesizes import A4        # dimensions du format A4
11#
12#     # 1) Choix d'un nom de fichier pour le document à produire :
13#     fichier = "document_1.pdf"
14#     # 2) Instanciation d'un "objet canevas" Reportlab lié à ce fichier :
15#     can = Canvas("{0}".format(fichier), pagesize=A4)
16#     # 3) Installation d'éléments divers dans le canevas :
17#     texte = "Mes œuvres complètes"                # ligne de texte à imprimer
18#     can.setFont("Times-Roman", 32)                 # choix d'une police de caractères
19#     posX, posY = 2.5*cm, 18*cm                     # emplacement sur la feuille
20#     can.drawString(posX, posY, texte)              # dessin du texte dans le canevas
21#     # 4) Sauvegarde du résultat dans le fichier PDF :
22#     can.save()

```

Après exécution de ce script, vous trouverez votre premier document PDF dans le répertoire courant, sous le nom de fichier **document\_1.pdf**. Il s'agira d'une page au format *DIN2 A4*, avec en son milieu la petite phrase « Mes œuvres complètes », reproduite en caractères *Times-Roman* de 24 points.

L'analyse de ce petit script vous montre qu'*avec ReportLab, vous disposez d'un langage de très haut niveau pour la description de pages imprimées*. Vous pourriez en effet condenser l'essentiel de son code en quatre lignes seulement, à l'aide de la composition d'instructions !<sup>3</sup>

### Commentaires

- ⑩ Ligne 5 : L'instruction `from __future__ import unicode_literals` en début de script force l'interpréteur Python 2 à convertir les chaînes littérales du script en chaînes Unicode (comme le fait Python 3). Sans cette instruction, il s'agirait de chaînes d'octets encodées suivant la norme utilisée par votre éditeur de textes<sup>4</sup>. Si cette norme est autre que Utf-8 (cas de Windows XP, par exemple), ces chaînes ne pourraient pas être acceptées par la bibliothèque *ReportLab*. Celle-ci n'accepte en effet que des chaînes Unicode ou des chaînes d'octets encodés en Utf-8 (indifféremment).

<sup>2</sup>Le DIN (*Deutsches Institut für Normung*) est l'organisme national de normalisation allemand.

<sup>3</sup>en combinant les lignes 11 et 13 d'une part, et les lignes 15, 17, 18 d'autre part.

<sup>4</sup>Cf. page Erreur : source de la référence non trouvée : « Problèmes éventuels liés aux caractères accentués ».



- ⑩ Ligne 8 : La bibliothèque *ReportLab* est volumineuse. Nous n'en importerons donc que les éléments nécessaires au travail envisagé. L'un des plus importants est la classe **Canvas()**, qui est dotée d'une quantité impressionnante de méthodes permettant de disposer à peu près n'importe quoi sur la page à imprimer : fragments de texte, paragraphes, dessins vectoriels, images bitmap, etc.
- ⑩ Lignes 9-10 : Les importations effectuées ici sont de simples valeurs destinées à améliorer la lisibilité du code. L'unité de mesure dans *ReportLab* est le point typographique *pica*, qui vaut 1/72° de pouce, soit 0.353 mm. **A4** est un simple tuple (595.28, 841.89) exprimant les dimensions d'une page DIN A4 dans ces unités, et **cm** la valeur d'un centimètre (28.346), que nous utiliserons abondamment dans la suite de nos exemples, afin d'exprimer plus clairement les positions et dimensions sur la page.
- ⑩ Ligne 15 : Instanciation d'un objet canevas. Il faut fournir au constructeur le nom du fichier destiné à recevoir le document, éventuellement accompagné de quelques arguments optionnels. Le format de page est déjà A4 par défaut, mais il n'est pas mauvais de le spécifier de manière explicite.
- ⑩ Ligne 18 : La méthode **setFont()** sert à choisir une police d'écriture et sa taille. L'une des grandes forces du PDF réside dans le fait que tous les logiciels destinés à l'interpréter (*Acrobat Reader*, etc.) doivent être munis d'une bibliothèque de *glyphes* identiques pour les polices vectorielles suivantes : *Courier*, *Courier-Bold*, *Courier-BoldOblique*, *Courier-Oblique*, *Helvetica*, *Helvetica-Bold*, *Helvetica-BoldOblique*, *Helvetica-Oblique*, *Symbol*, *Times-Bold*, *Times-BoldItalic*, *Times-Italic*, *Times-Roman*, *ZapfDingbats*. Ces polices sont largement suffisantes pour une multitude d'usages : ce sont en effet une police à chasse fixe (*Courier*), deux polices proportionnelles avec et sans empattements (*Times-Roman*, *Helvetica*), chacune d'elles dans quatre variantes de style (normal, gras, italique, gras & italique) et enfin deux polices de symboles et pictogrammes divers (*Symbol*, *ZapfDingbats*). Le fait qu'elles soient déjà « connues » du logiciel interpréteur vous dispense d'avoir à les décrire dans le document lui-même : si vous pouvez vous contenter de ces polices, vos documents PDF conserveront une taille très modeste.

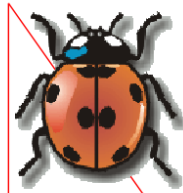
#### **PERFORMANCES Embarquer d'autres polices vectorielles ?**

*Il est parfaitement possible d'utiliser d'autres polices vectorielles (TrueType ou Adobe Type 1), mais alors il faut en fournir la description numérique dans le document lui-même, ce qui augmentera fortement la taille de celui-ci et compliquera un peu les choses. Nous n'expliquerons pas la marche à suivre dans cet ouvrage.*

- ⑩ Ligne 20 : La méthode **drawString(x, y, t)** positionne le fragment de texte **t** dans le canevas en l'alignant à gauche sur le point de coordonnées **x, y**. Il est très important de signaler ici que ces coordonnées doivent être comptées à partir du **coin inférieur gauche** de la page, comme le veut l'usage en mathématiques, et non à partir du coin supérieur gauche comme sont habituellement comptées les coordonnées d'écran. Si vous voulez écrire plusieurs lignes de texte successives dans la page, vous devrez donc faire en sorte que leur coordonnée verticale **diminue** de la première à la dernière.
- ⑩ Ligne 20 : La méthode **save()** finalise le travail et referme le fichier correspondant. Nous verrons plus loin comment générer des documents de plusieurs pages.

### **Générer un document plus élaboré**

Le script suivant génère un document un peu bizarre, pour mettre en évidence quelques-unes des nombreuses possibilités qui sont désormais à notre portée, dont celle de reproduire des images bitmap sur la page. Si vous souhaitez utiliser cette fonctionnalité, vous devez cependant veiller à ajouter à votre installation de Python la bibliothèque de traitement d'images **Python Imaging Library (PIL)**, outil remarquable qui pourrait faire l'objet d'un livre à lui tout seul. Notez au passage que les documentations détaillées de *ReportLab* et de *PIL* sont disponibles sur le Web, tout au moins en anglais. L'installation de *PIL* est décrite page **Erreur ! Signet non défini.**



*L'espoir fait vivre.*

**Petite pluie abat grand vent.**

Qui ne risque rien, n'a rien.

La nuit porte conseil.



*L'exception confirme la règle*

```

1# # === Génération d'un document PDF avec divers types de tracés simples ===
2#
3# # Adaptations du script pour le rendre exécutable sous Python 2.6 ou 2.7 :
4# # (Ces lignes peuvent être supprimées si Reportlab est disponible pour Python3)
5# from __future__ import unicode_literals
6# from __future__ import division # division "réelle"
7# # -----
8#
9# # Importation de quelques éléments de la bibliothèque ReportLab :
10# from reportlab.pdfgen.canvas import Canvas
11# from reportlab.lib.units import cm
12# from reportlab.lib.pagesizes import A4
13#
14# fichier = "document_2.pdf"
15# can = Canvas("{0}".format(fichier), pagesize=A4)
16# # Installation d'éléments divers dans le canevas :
17# largeurP, hauteurP = A4 # largeur et hauteur de la page
18# centreX, centreY = largeurP/2, hauteurP/2 # coordonnées du centre de la page
19# can.setStrokeColor("red") # couleur des lignes
20# # Rappel : la position verticale sur la page est comptée à partir du bas.
21# can.line(1*cm, 1*cm, 1*cm, 28*cm) # ligne verticale à gauche
22# can.line(1*cm, 1*cm, 20*cm, 1*cm) # ligne horizontale en bas
23# can.line(1*cm, 28*cm, 20*cm, 1*cm) # ligne oblique (descendante)
24# can.setLineWidth(3) # nouvelle épaisseur des lignes
25# can.setFillColorRGB(1, 1, .5) # couleur de remplissage (RVB)
26# can.rect(2*cm, 2*cm, 18*cm, 20*cm, fill=1) # rectangle de 18 x 20 cm
27#
28# # Dessin d'un bitmap (aligné par son coin inférieur gauche). La méthode
29# # drawImage renvoie les dimensions du bitmap (en pixels) dans un tuple :
30# dX, dY = can.drawImage("cocci3.gif", 1*cm, 23*cm, mask="auto")
31# ratio = dY/dX # rapport haut./larg. de l'image
32# can.drawImage("cocci3.gif", 1*cm, 14*cm,
33# width=3*cm, height=3*cm*ratio, mask="auto")
34# can.drawImage("cocci3.gif", 1*cm, 7*cm, width=12*cm, height=5*cm, mask="auto")
35#
36# can.setFillColorCMYK(.7, 0, .5, 0) # couleur de remplissage (CMJN)
37# can.ellipse(3*cm, 4*cm, 19*cm, 10*cm, fill=1) # ellipse (! axes = 16 x 6 cm)
38# can.setLineWidth(1) # nouvelle épaisseur des lignes
39# can.ellipse(centreX -.5*cm, centreY -.5*cm, # petit cercle indiquant la
40# centreX +.5*cm, centreY +.5*cm) # position du centre de la page
41#
42# # Quelques textes, avec polices, couleurs, orientations et alignements divers :
43# can.setFillColor("navy") # couleur des textes
44# texteC = "Petite pluie abat grand vent." # texte à centrer
45# can.setFont("Times-Bold", 18)
46# can.drawCentredString(centreX, centreY, texteC)
47# texteG = "Qui ne risque rien, n'a rien." # texte à aligner à gauche
48# can.setFont("Helvetica", 18)
49# can.drawString(centreX, centreY -1*cm, texteG)
50# texteD = "La nuit porte conseil." # texte à aligner à droite
51# can.setFont("Courier", 18)
52# can.drawRightString(centreX, centreY -2*cm, texteD)
53# texteV = "L'espoir fait vivre." # texte à disposer verticalement
54# can.rotate(90)
55# can.setFont("Times-Italic", 18)
56# can.drawString(centreY +1*cm, -centreX, texteV) # ! inversion des coordonnées !
57# texteE = "L'exception confirme la règle" # texte à afficher en blanc
58# can.rotate(-90) # retour à l'orientation horiz.
59# can.setFont("Times-BoldItalic", 28)
60# can.setFillColor("white") # nouvelle couleur des textes
61# can.drawCentredString(centreX, 7*cm, texteE)
62#
63# can.save() # Sauvegarde du résultat

```

## Commentaires

⑩ Ligne 6 : Sous Python 2, l'opérateur de division `/` effectue une division entière par défaut (correspondant à l'opérateur `//` sous Python 3 – voyez page **Erreur ! Signet non défini.**). On force ici le mode de division réel.

⑩ Lignes 17 à 40 : Ces lignes vous montrent quelques-unes des méthodes permettant de tracer des dessins sur la page à l'aide de lignes et de formes basiques. Bien évidemment, nous ne pouvons vous présenter ici que les plus simples. Si vous prenez la peine de consulter la documentation de référence de *ReportLab* (soit plusieurs manuels en PDF !), vous y trouverez une quantité phénoménale d'autres méthodes permettant de réaliser des dessins vectoriels redimensionnables, des graphiques, des tableaux, des histogrammes, des diagrammes circulaires, etc.

⑩ Les lignes 19 à 23 définissent un triangle qui se résume à un contour rouge. Les lignes 24 à 26 dessinent un rectangle à contour plus épais et rempli d'une couleur jaune pâle. Remarquez que l'ordre des instructions est déterminant : les dessins tracés successivement se superposent. Notez bien encore une fois que toutes les coordonnées verticales sont comptées vers le haut, à partir du bas de la page.

⑩ Lignes 19, 25, 36, 43, 60 : Dans *ReportLab*, les couleurs peuvent être désignées de trois manières différentes. La plus simple consiste à utiliser des noms de couleurs en anglais, mais cela ne convient guère pour choisir des nuances précises. On peut alors désigner celles-ci en indiquant leurs 3 composantes de lumière rouge, verte et bleue (*RVB*, ou *RGB en anglais*) comme on le fait pour les pixels d'un écran, ou mieux encore, puisqu'il s'agit d'une page imprimée, en indiquant les 4 composantes Cyan, Magenta, Jaune et Noir (*CMJN*, ou *CMYK en anglais*) des encres à utiliser pour reproduire l'image sur papier en quadrichromie. Les valeurs de chaque composante doivent être comprises entre zéro et un.

⑩ Lignes 28 à 34 : La méthode **drawImage()** est utilisée ici pour reproduire trois fois la même image de coccinelle, à des emplacements différents et redimensionnée. Ceci est rendu possible grâce aux puissantes fonctions de la *PIL*. Notez que si une même image est utilisée plusieurs fois dans le même document, elle n'est cependant chargée qu'une seule fois dans le PDF, via un mécanisme de cache.

⑩ Ligne 30 : On reproduit une première fois l'image bitmap sans la redimensionner. *ReportLab* considère dans ce cas que chaque pixel de l'image correspond à un point *pica* ( $1/72^{\circ}$  de pouce). Le premier argument transmis à la méthode **drawImage()** est le nom de fichier contenant l'image. Les formats acceptés sont *PNG*, *JPG*, *GIF*, *TIF* et *BMP*. Les deuxième et troisième arguments obligatoires sont les coordonnées de son coin inférieur gauche sur la page. L'argument optionnel **mask="auto"** est requis si vous souhaitez que la transparence éventuelle de l'image soit respectée<sup>5</sup>.

⑩ Lignes 31 à 33 : Une particularité utile de la méthode **drawImage()** est qu'elle renvoie les dimensions du bitmap en pixels, dans un tuple d'entiers. Vous pouvez donc utiliser cette information dans votre script, comme ici pour déterminer le rapport hauteur/largeur de l'image, afin de pouvoir la redessiner ailleurs à une autre échelle, grâce aux paramètres optionnels **width** et **height**.

⑩ Lignes 34 à 40 : On redessine encore la même image, cette fois avec des dimensions quelconques, puis on la recouvre partiellement avec une ellipse. Notez qu'à la différence de la méthode **rect()** de la ligne 26, la méthode **ellipse()** attend 4 arguments obligatoires qui sont les coordonnées x, y des coins inférieur gauche et supérieur droit du rectangle dans lequel cette ellipse doit s'inscrire, alors que pour la méthode **rect()**, ces 4 arguments sont d'abord les deux coordonnées x, y du coin inférieur gauche du rectangle, suivis de sa largeur et de sa hauteur. Les lignes 38 à 40 dessinent un petit cercle au centre de la page, lequel nous servira de repère pour comprendre le positionnement des quelques lignes de texte tracées par nos dernières instructions.

⑩ Lignes 42 à 61 : Veuillez examiner attentivement ces lignes. Nous y montrons comment vous pouvez aligner à gauche, aligner à droite, ou centrer une ligne de texte, à l'aide des méthodes **drawString()**, **drawRightString()** et **drawCentredString()**. Vous pouvez bien entendu utiliser différentes polices, couleurs et tailles de caractères, et même faire tourner votre texte sous un angle quelconque !

<sup>5</sup>Ce « masque » peut contenir d'autres valeurs, mais il est clair que dans cette présentation forcément très limitée des potentialités de *ReportLab*, nous ne pouvons pas nous permettre de détailler les variantes de chaque option ou argument présentés.



## Gestion des paragraphes avec ReportLab

La **programmation** est l'art d'apprendre à une machine comment accomplir de nouvelles tâches, qu'elle n'avait jamais été capable d'effectuer auparavant.

C'est par la programmation que vous pourrez acquérir le plus de contrôle, non seulement sur votre machine, mais aussi peut-être sur celles des autres par l'intermédiaire des réseaux. D'une certaine façon, cette activité peut donc être assimilée à une forme particulière de magie.

Elle donne effectivement à celui qui l'exerce un certain pouvoir, mystérieux pour le plus grand nombre, voire inquiétant quand on se rend compte qu'il peut être utilisé à des fins malhonnêtes.



Dans le monde de la programmation, on désigne par le terme **hacker** les programmeurs chevronnés qui ont perfectionné les systèmes d'exploitation de type Unix et mis au point les techniques de communication qui sont à la base du développement extraordinaire de l'Internet.

Ce sont eux également qui continuent inlassablement à produire et à améliorer les logiciels libres (*Open Source*).



Selon notre analogie, les hackers sont donc des maîtres-sorciers, qui pratiquent la magie blanche.

Mais il existe aussi un autre groupe de gens que les journalistes mal informés désignent erronément sous le nom de *hackers*, alors qu'ils devraient plutôt les appeler *crackers*.

Ces personnes se prétendent *hackers* parce qu'ils veulent faire croire qu'ils sont très compétents, alors qu'en général ils ne le sont guère.

Ils sont cependant très nuisibles, parce qu'ils utilisent leurs quelques connaissances pour rechercher les moindres failles des systèmes informatiques construits par d'autres, afin d'y effectuer toutes sortes d'opérations illicites : vol d'informations confidentielles, escroquerie, diffusion de spam, de virus, de propagande haineuse, de pornographie et de contrefaçons, destruction de sites web, etc.

Ces sorciers dépravés s'adonnent bien sûr à une forme grave de magie noire.



Mais il y en a une autre.

Les vrais *hackers* cherchent à promouvoir dans leur domaine une certaine éthique, basée principalement sur l'émulation et le partage des connaissances. La plupart d'entre eux sont des perfectionnistes, qui veillent non seulement à ce que leurs constructions logiques soient efficaces, mais aussi à ce qu'elles soient élégantes, avec une structure parfaitement lisible et documentée.

Vous découvrirez rapidement qu'il est aisé de produire à la va-vite des programmes qui fonctionnent, certes, mais qui sont obscurs et confus, indéchiffrables pour toute autre personne que leur auteur (et encore !).

Cette forme de programmation absconse et ingérable est souvent aussi qualifiée de « magie noire » par les *hackers*.

#### La démarche du programmeur

Comme le sorcier, le programmeur compétent semble doté d'un pouvoir étrange qui lui permet de transformer une machine en une autre, une machine à calculer en une machine à écrire ou à dessiner, par exemple, un peu à la manière d'un sorcier qui transformerait un prince charmant en grenouille, à l'aide de quelques incantations mystérieuses entrées au clavier.

Comme le sorcier, il est capable de guérir une application apparemment malade, ou de jeter des sorts à d'autres, via l'Internet.

Mais comment cela est-il possible ?

## Documents de plusieurs pages et gestion des paragraphes

Dans ce chapitre d'initiation aux fonctionnalités de *ReportLab*, nous ne pouvons qu'effleurer un sujet en réalité très vaste. Ce que nous avons expliqué sommairement dans les pages précédentes ne concerne que la couche logicielle la plus basse de cette bibliothèque, le niveau « *canevas* ». Au-dessus de cette première couche, il en existe encore quatre autres :

- ⑩ les éléments « *fluables6* » : dont les plus importants sont les *paragraphes* (portions de texte formatées), mais qui peuvent être aussi des images, des espacements, des tables, etc., et qui ont en commun de pouvoir être « coulés » les uns à la suite des autres dans des régions prédéfinies du document ;

<sup>6</sup>Fluable : « qui coule, qui est liquide » (du latin *fluere*). Adjectif très peu utilisé dans le langage courant, utilisé ici comme traduction approximative du néologisme anglais *flowable* formé à partir du verbe *to flow* (s'écouler, fluier) et qui signifie

- ⑩ les *cadres*, qui définissent donc ces régions rectangulaires sur une page, où l'on pourra « couler » les différents éléments *fluables* décrits ci-dessus, en un flux continu ;
- ⑩ les *styles de pages*, qui constituent différents modèles de pages avec des en-têtes, bas de pages, cadres et numérotations déjà en place ;
- ⑩ les *styles de documents (ou modèles)*, qui proposent différentes mises en page prédéfinies.

À la lecture de ce qui précède, vous aurez compris que ce qui est mis à votre disposition avec *ReportLab* est bien un outil professionnel de très haut niveau, qui offre tout ce que l'on peut attendre d'un système de traitement de texte moderne. La place nous manque évidemment pour vous en fournir une description complète, mais nous allons tenter d'en expliquer les principaux mécanismes mis en œuvre aux niveaux 2 et 3, à savoir les *fluables* et les *cadres*.

*Nos explications devraient vous suffire pour préparer vos premiers documents, mais nous ne saurions trop vous encourager à consulter aussi l'abondante documentation disponible en ligne sur le site web de ReportLab, afin d'en tirer un maximum de profit.*

### Exemple de script pour la mise en page d'un fichier texte

Le script ci-après charge les lignes d'un simple fichier texte dans une liste. Chaque ligne est ensuite transformée en objet *fluable* de type « paragraphe ». D'autres *fluables* sont générés et intercalés entre ces paragraphes : un espacement après chacun d'eux, et de temps à autre une image formatée. Tous ces objets sont placés dans une liste unique, qui sert ensuite de source de flux pour alimenter le remplissage de cinq cadres (*frames*). Le surplus disponible à la fin est traité différemment, afin de mettre en lumière quelques méthodes plus basiques des objets-paragraphe.

```

1# # == Génération d'un document PDF avec gestion de fluables (paragraphes) ==
2#
3# # Adaptations du script pour le rendre exécutable sous Python 2.6 ou 2.7 :
4# # (Ces lignes peuvent être supprimées si Reportlab est disponible pour Python3)
5# from __future__ import unicode_literals
6# from __future__ import division # division "réelle"
7# from codecs import open # décodage des fichiers texte
8# -----
9#
10# # Importer quelques éléments de la bibliothèque ReportLab :
11# from reportlab.pdfgen.canvas import Canvas
12# from reportlab.lib.units import cm
13# from reportlab.lib.pagesizes import A4
14# from reportlab.platypus import Paragraph, Frame, Spacer
15# from reportlab.platypus.flowables import Image as rImage
16# from reportlab.lib.styles import getSampleStyleSheet
17#
18# # Créer une liste des chaînes de caractères à convertir + loin en paragraphes :
19# ofi = open("document.txt", "r", encoding="Utf8")
20# txtList = []
21# while 1:
22#     ligne = ofi.readline()
23#     if not ligne:
24#         break
25#     txtList.append(ligne)
26# ofi.close()
27#
28# # == Construction du document PDF :
29# fichier = "document_3.pdf"
30# can = Canvas("{0}".format(fichier), pagesize=A4)
31# styles = getSampleStyleSheet() # dictionnaire de styles prédéfinis
32# styleN = styles["Normal"] # objet de classe ParagraphStyle()
33#
34# # Les paragraphes, interlignes et figures seront appelés éléments "fluables".
35# # Insertion de ces éléments fluables dans la liste <story> ("l'histoire") :
36# n, f, story = 0, 0, []

```

ici « élément d'un flux d'entités imprimables ».

```

37# for txt in txtList:
38#     story.append(Paragraph(txt, styleN)) # ajouter un paragraphe
39#     n +=1 # compter les paragraphes générés
40#     story.append(Spacer(1, .2*cm)) # ajouter un espacement (de 2mm)
41#     f +=2 # compter les fluables générés
42#     if n in (3,5,10,18,27,31): # ajouter une image bitmap
43#         story.append(rllimage("cocci3.gif", 3*cm, 3*cm, kind="proportional"))
44#         f +=1
45#
46# # == Préparation de la première page :
47# can.setFont("Times-Bold", 18)
48# can.drawString(5*cm, 28*cm, "Gestion des paragraphes avec ReportLab")
49# # Mise en place de trois cadres (2 "colonnes" et un "bas de page") :
50# cG =Frame(1*cm, 11*cm, 9*cm, 16*cm, showBoundary =1)
51# cD =Frame(11*cm, 11*cm, 9*cm, 16*cm, showBoundary =1)
52# cI =Frame(1*cm, 3*cm, 19*cm, 7*cm, showBoundary =1)
53# # Mise en place des éléments fluables dans ces trois cadres :
54# cG.addFromList(story, can) # remplir le cadre de gauche
55# cD.addFromList(story, can) # remplir le cadre de droite
56# cI.addFromList(story, can) # remplir le cadre inférieur
57#
58# can.showPage() # passer à la page suivante
59#
60# # == Préparation de la deuxième page :
61# cG =Frame(1*cm, 12*cm, 9*cm, 15*cm, showBoundary =1) # deux cadres
62# cD =Frame(11*cm, 12*cm, 9*cm, 15*cm, showBoundary =1) # (= 2 colonnes)
63# cG.addFromList(story, can) # remplir le cadre de gauche
64# cD.addFromList(story, can) # remplir le cadre de droite
65#
66# # Traitement individuel des éléments fluables restants :
67# xPos, yPos = 6*cm, 11.5*cm # position de départ
68# lDisp, hDisp = 14*cm, 14*cm # largeur et hauteur disponibles
69# for flua in story:
70#     f += 1
71#     l, h =flua.wrap(lDisp, hDisp) # largeur et hauteur effectives
72#     if flua.identity()[1:10] == "Paragraph":
73#         can.drawString(2*cm, yPos-12, "Fluable n° {0}".format(f))
74#         flua.drawOn(can, xPos, yPos-h) # installer le fluable
75#         yPos -=h # position du suivant
76#
77# can.save() # finaliser le document

```

## Commentaires

⑩ Ligne 14 : ReportLab fournit une classe de cadres (*frames*), et plusieurs classes d'entités fluables pouvant y être « coulées » en un flux continu. Nous n'utiliserons ici que les fluables des types *paragraphe*, *espacement* et *image*. Notons au passage l'utilisation de **as** pour renommer la classe **Image()** importée : c'est une précaution utile pour éviter la confusion avec la classe **Image()** de la *Python Imaging Library* (PIL) qui pourrait éventuellement être utilisée aussi dans notre script.

⑩ Ligne 16 : La classe **Paragraph()** est très élaborée : la documentation de ReportLab y consacre tout un chapitre. Les objets instanciés à partir de cette classe permettent une mise en page précise de portions de textes quelconques, formatés selon vos souhaits dans un certain style. Quelques styles de base sont disponibles dans le module **reportlab.lib.styles**, dont l'attribut **getSampleStyleSheet** est organisé comme un dictionnaire, mais l'important est de savoir que vous pouvez aisément modifier n'importe lequel d'entre eux pour l'adapter à vos besoins en modifiant ses différents attributs par défaut (police de caractères, couleur, valeurs des retraits et/ou des espacements, tabulations, ajout de puces ou de numéros, alignements, etc.). Un exemple de modification d'un style de paragraphe est proposé plus loin, dans les exercices à la fin de ce chapitre.

⑩ Lignes 18 à 26 : La source de nos paragraphes est un simple fichier texte, dont nous extrayons les chaînes de caractères de la manière habituelle (cf. page **Erreur ! Signet non défini.**). Veuillez remarquer que ce texte peut contenir un certain nombre de balises de formatage en ligne de type XML, comme **<u>** et **</u>** pour délimiter une portion de texte à souligner, par exemple, ou **<i>** et **</i>** pour la présenter en italique, etc.). Nous ne pouvons évidemment pas fournir ici la liste de toutes ces balises. Sachez cependant que cela



entraîne la conséquence que vous ne pouvez pas utiliser certains caractères réservés tels que `<` et `&` dans un texte destiné à être transformé en paragraphe ReportLab.

⑩ Ligne 32 : Cette instruction place dans la variable `styleN` un objet de la classe `ParagraphStyle()`, définissant le style choisi pour tous nos paragraphes, en l'occurrence un style de texte courant en police *Times-Roman*. Comme déjà signalé plus haut, nous pourrions aisément modifier ce style en changeant la valeur par défaut de ses différents attributs (cf. exercices en fin de chapitre).

⑩ Lignes 34 à 44 : C'est ici que nous construisons la liste des objets fluables qui seront coulés plus loin, en un flux continu, dans les différents cadres préparés pour eux. Cette liste sera en quelque sorte une « histoire » que nous allons raconter en mettant bout à bout des fragments de texte, des images, etc. dans différents espaces réservés pour elle sur nos pages. Cela explique le choix du nom de variable `story` habituellement choisi pour désigner cette liste. Pour chaque ligne extraite du fichier texte, nous lui ajoutons un objet fluable de type paragraphe, instancié à la ligne 38 avec le style défini dans `styleN`, immédiatement suivi d'un fluable de type espacement, instancié à la ligne 40. Après certains paragraphes, nous intercalerons aussi quelques fluables de type image (lignes 42-43). Ces images peuvent être redimensionnées à volonté (l'argument optionnel `kind="proportional"` force la conservation du ratio d'aspect (c'est-à-dire le rapport hauteur/largeur ; les dimensions qui le précèdent sont alors des maxima). Accessoirement, nous effectuons en parallèle un comptage des paragraphes et des fluables dans les variables `n` et `f`, mais ce n'est évidemment pas indispensable.

⑩ Lignes 49 à 52 : Sur la première page de notre document, nousinstancions trois cadres (objets de la classe `Frame()` de ReportLab) : deux colonnes verticales surmontant un rectangle horizontal. Ces cadres seront donc les espaces où viendront « s'écouler » nos fluables. Pour chaque cadre, il faut fournir dans l'ordre : les coordonnées du coin inférieur gauche du rectangle (comptées à partir du bas de la page), sa largeur et sa hauteur. Afin que le code reste bien lisible, nous exprimons les dimensions en centimètres, et les convertissons en points *pica* grâce à l'utilisation de la « constante » `cm`. L'argument optionnel `showBoundary=1` permet de visualiser effectivement les cadres rectangulaires, pendant les phases de développement de vos scripts. Lorsque ceux-ci seront au point, il vous suffira de supprimer cet argument (ou de lui donner une valeur nulle) pour les faire disparaître.

- ⑩ Lignes 54 à 56 : La méthode `addFromList()` des objets cadres créés à l'étape précédente effectue le

 <p>Cela peut paraître paradoxal, mais comme nous l'avons déjà fait remarquer plus haut, le vrai maître est en fait celui qui ne croit à aucune magie, à aucun don, à aucune intervention surnaturelle.</p> <p>Seule la froide, l'implacable, l'inconfortable logique est de mise.</p> <p>Le mode de pensée d'un programmeur combine des constructions intellectuelles complexes, similaires à celles qu'accomplissent les mathématiciens, les ingénieurs et les scientifiques.</p> <p>Comme le mathématicien, il utilise des langages formels pour décrire des raisonnements (ou algorithmes). Comme l'ingénieur, il conçoit des dispositifs, il assemble des composants pour réaliser des mécanismes et il évalue leurs performances. Comme le scientifique, il observe le comportement de systèmes complexes, il crée des modèles, il teste des prédictions.</p> <p><i>L'activité essentielle d'un programmeur consiste à résoudre des problèmes.</i></p>	<p>Il s'agit-là d'une compétence de haut niveau, qui implique des capacités et des connaissances diverses : être capable de (re)formuler un problème de plusieurs manières différentes, être capable d'imaginer des solutions innovantes et efficaces, être capable d'exprimer ces solutions de manière claire et complète.</p> <p>Comme nous l'avons déjà évoqué plus haut, il s'agit souvent de mettre en lumière les implications concrètes d'une représentation mentale « magique », simpliste ou trop abstraite.</p> <p>La programmation d'un ordinateur consiste en effet à « expliquer » en détail à une machine ce qu'elle doit faire, en sachant d'emblée qu'elle ne peut pas véritablement « comprendre » un langage humain, mais seulement effectuer un traitement automatique sur des séquences de caractères.</p> <p>Il s'agit là plupart du temps de convertir un souhait exprimé à l'origine en termes « magiques », en un vrai raisonnement parfaitement structuré et élucidé dans ses moindres détails, que l'on appelle un algorithme.</p>  <p>Considérons par exemple une suite de nombres fournis dans le désordre : 47, 19, 23, 15, 21, 36, 5, 12 ...</p> <p>Comment devons-nous nous y prendre pour obtenir d'un ordinateur qu'il les remette dans l'ordre ?</p>
Fluable n° 77	Le souhait « magique » est de n'avoir qu'à cliquer sur un bouton, ou entrer une seule instruction au clavier, pour qu'automatiquement les nombres se mettent en place. Mais le travail du sorcier-programmeur est justement de créer cette « magie ».
Fluable n° 79	Pour y arriver, il devra décortiquer tout ce qu'implique pour nous une telle opération de tri (au fait, existe-t-il une méthode unique pour cela, ou bien y en a-t-il plusieurs ?), et en traduire toutes les étapes en une suite d'instructions simples, telles que par exemple « comparer les deux premiers nombres, les échanger s'ils ne sont pas dans l'ordre souhaité, recommencer avec le deuxième et le troisième, etc. ».
Fluable n° 82	Si les instructions ainsi mises en lumière sont suffisamment simples, il pourra alors les encoder dans la machine en respectant de manière très stricte un ensemble de conventions fixées à l'avance, que l'on appelle un langage informatique.
Fluable n° 84	Pour « comprendre » celui-ci, la machine sera pourvue d'un mécanisme qui décode ces instructions en associant à chaque « mot » du langage une action précise.
Fluable n° 86	Ainsi seulement, la magie pourra s'accomplir.
Fluable n° 88	

travail de remplissage, à partir de la liste `story`. Elle en extrait les *fluables* un à un et les installe l'un en-dessous de l'autre dans le cadre, jusqu'à ce que celui-ci soit rempli, en effectuant automatiquement les sauts à la ligne nécessaires pour éviter la coupure des mots, et en appliquant toutes les indications de styles choisies. Lorsque le cadre est rempli, la liste résiduelle dans `story` peut être utilisée pour remplir d'autres cadres, et ainsi de suite.

*Notez que si un paragraphe est trop grand pour pouvoir être installé dans le cadre, une exception est générée. On peut détecter celle-ci pour provoquer une division automatique du paragraphe en paragraphes plus petits. Consultez la documentation de ReportLab pour plus de détails.*

- ⑩ La ligne 58 clôture la page courante et provoque l'insertion d'une nouvelle page dans le document.
- ⑩ Lignes 60 à 64 : La page nouvellement insérée est vierge. On y installe d'autres cadres, et on les remplit avec les *fluables* que l'on continue à extraire de la liste `story`, mais à la fin de l'opération celle-ci n'est pas encore vide : nous allons nous servir des *fluables* restants pour vous en expliquer un traitement plus basique, permettant un contrôle plus fin de leur mise en place sur la page.
- ⑩ Lignes 67-68 : Nous choisissons un point de départ sur la page (toujours compté à partir du bas !), et les dimensions *maximales* (largeur et hauteur) que nous décidons d'allouer à chacun de nos *fluables*. En l'occurrence, nous choisissons un carré de 14x14 cm. De ces deux valeurs, seule la largeur sera limitante, car aucun des *fluables* restants n'est suffisamment grand pour exiger une hauteur de 14 cm.



⑩ Lignes 69 à 75 : Parcours de la liste des *fluables* restants. Au lieu de les « écouler » dans des cadres rigides prédéfinis comme nous l'avons fait jusqu'à présent, nous allons cette fois les positionner directement sur le canevas, en déterminant l'espace exact nécessaire pour chacun d'eux. La méthode `wrap()` de l'objet *fluable* en cours de traitement est destinée à cet usage. Elle prend en arguments les dimensions maximales de l'espace que l'on veut allouer au *fluable* (c'est-à-dire en général : la largeur qu'on veut lui voir prendre, et une hauteur nettement trop importante), et elle renvoie en retour la largeur et la hauteur qui seront effectivement utilisées. Dans notre exemple où nous avons choisi une largeur fixe, la hauteur ainsi trouvée permet de positionner les *fluables* les uns en-dessous des autres (méthode `drawOn()`, ligne 74) en gardant la connaissance de leurs coordonnées effectives à tout moment (c'est ce qui nous permet par exemple de positionner un libellé en regard de chaque paragraphe (ligne 73). La méthode `identity()` utilisée à la ligne 72 permet accessoirement de déterminer quel est le type (paragraphe, image, espacement) du *fluable* en cours de traitement.

### En conclusion

Croyez bien que nous n'avons exposé ici qu'un aperçu très restreint de l'énorme potentiel de ressources que représente la bibliothèque ReportLab. Il est clair que nous avons passé sous silence un grand nombre de concepts et de méthodes. Nous n'avons pas indiqué, par exemple, ce qu'il faut faire pour sectionner un paragraphe trop grand, créer des tableaux ou des graphiques, des modèles de documents, des hyperliens, etc., ni comment crypter nos PDF ou y inclure des formulaires, des effets de transitions de page, etc. Encore une fois, nous ne saurions trop vous recommander de consulter l'abondante documentation disponible en ligne pour cette bibliothèque, ainsi que celle de la *Python Imaging Library*.